A Pseudocodes

A.1 Global Contribution Partial Updating

The magnitude pruning method prunes (*i.e.*, set as zero) weights with the lowest magnitudes in a network, which is the current best-performed pruning method aiming at the trade-off between the model accuracy and the number of zero's weights (Renda et al., 2020). We adapt the magnitude pruning proposed in (Renda et al., 2020) to prune the incremental weights $\delta\boldsymbol{w}^{\mathrm{f}}$. Specially, the elements with the smallest absolute values in $\delta\boldsymbol{w}^{\mathrm{f}}$ are set to zero (also rewinding), while the remaining weights are further sparsely fine-tuned with the same learning rate schedule as training $\boldsymbol{w}^{\mathrm{f}}$.

Algorithm 1: Global Contribution Partial Updating (Prune Incremental Weights)

```
Input: weights $\boldsymbol{w}$, dataset $\mathcal{D}$, updating ratio $k$, learning rate $\{\alpha^q\}_{q=1}^{Q}$ in $Q$ iterations
Output: weights $\widetilde{\boldsymbol{w}}$
/* The first step:  full updating and rewinding (pruning)     */
Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$;
for $q \leftarrow 1$ to $Q$ do
    Compute the loss gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \frac{\partial \ell(\boldsymbol{w}^{q-1})}{\partial \boldsymbol{w}^{q-1}}$;
    Compute the optimization step with learning rate $\alpha^q$ as $\Delta\boldsymbol{w}^q$;
    Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta\boldsymbol{w}^q$;
Get $\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^Q$;
Compute the increment of weights $\delta\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^{\mathrm{f}} - \boldsymbol{w}$;
Compute $\boldsymbol{c}^{\mathrm{global}} = \delta\boldsymbol{w}^{\mathrm{f}} \odot \delta\boldsymbol{w}^{\mathrm{f}}$;
Sort the values in $\boldsymbol{c}^{\mathrm{global}}$ in descending order;
Create a mask $\boldsymbol{m}$ with $1$ for the indices of Top-$k \cdot I$ values in the above order, $0$ for others;
/* The second step:  sparse fine-tuning                         */
Initiate $\widetilde{\delta\boldsymbol{w}} = \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$;
Initiate $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$;
for $q \leftarrow 1$ to $Q$ do
    Compute the optimization step on $\widetilde{\boldsymbol{w}}$ with learning rate $\alpha^q$ as $\Delta\widetilde{\boldsymbol{w}}^q$;
    Update $\widetilde{\delta\boldsymbol{w}} = \widetilde{\delta\boldsymbol{w}} + \Delta\widetilde{\boldsymbol{w}}^q \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$;
```

In comparison to traditional pruning on weights, pruning on incremental weights has a different start point. Traditional pruning on weights first trains randomly initialized weights (a zero-initialized network cannot be trained due to the symmetry), and then prunes the weights with the smallest magnitudes. However, the increment of weights $\delta\boldsymbol{w}^{\mathrm{f}}$ is initialized with zero in Alg. 1, since the first step starts from $\boldsymbol{w}$. This implies that pruning $\delta\boldsymbol{w}^{\mathrm{f}}$ has the same functionality as rewinding these weights to their initial values in $\boldsymbol{w}$.

A.2 Deep Partial Updating

Alg. 2 presents the pseudocode of our deep partial updating method, namely rewinding according to the combined contribution (the sum of normalized global contribution and local contribution) w.r.t. the loss reduction.

B Complexity Analysis

Algorithm 1: Global Contribution Partial Updating. Recall that the dimensionality of the weights vector is denoted as I. In Q optimization iterations during the first step, Alg. 1 does not introduce extra time complexity or extra space complexity related to the original optimizer. The rest of the first step takes a time complexity of $O(I \cdot \log(I))$ and a space complexity of $O(I)$, (*e.g.*, using heap sort or quick sort). In Q optimization iterations during the second step, Alg. 1 introduces an extra time

Algorithm 2: Deep Partial Updating

```
Input: weights $\boldsymbol{w}$, dataset $\mathcal{D}$, updating ratio $k$, learning rate $\{\alpha^q\}_{q=1}^Q$ in $Q$ iterations
Output: weights $\widetilde{\boldsymbol{w}}$
/* The first step: full updating and rewinding */
Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$;
Initiate $\boldsymbol{c}^{\mathrm{local}} = \mathbf{0}$;
for $q \leftarrow 1$ to $Q$ do
    Compute the loss gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \frac{\partial \ell(\boldsymbol{w}^{q-1})}{\partial \boldsymbol{w}^{q-1}}$;
    Compute the optimization step with learning rate $\alpha^q$ as $\Delta \boldsymbol{w}^q$;
    Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta \boldsymbol{w}^q$;
    Update $\boldsymbol{c}^{\mathrm{local}} = \boldsymbol{c}^{\mathrm{local}} - \boldsymbol{g}(\boldsymbol{w}^{q-1}) \odot \Delta \boldsymbol{w}^q$;
Get $\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^Q$;
Compute the increment of weights $\delta \boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^{\mathrm{f}} - \boldsymbol{w}$;
Compute $\boldsymbol{c}^{\mathrm{global}} = \delta \boldsymbol{w}^{\mathrm{f}} \odot \delta \boldsymbol{w}^{\mathrm{f}}$;
Compute the combined contribution $\boldsymbol{c} = \boldsymbol{c}^{\mathrm{global}} / (\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{global}}) + \boldsymbol{c}^{\mathrm{local}} / (\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{local}})$;
Sort the values in $\boldsymbol{c}$ in descending order;
Create a mask $\boldsymbol{m}$ with $1$ for the indices of Top-$k \cdot I$ values in the above order, $0$ for others;
/* The second step: sparse fine-tuning */
Initiate $\widetilde{\delta \boldsymbol{w}} = \delta \boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$;
Initiate $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta \boldsymbol{w}}$;
for $q \leftarrow 1$ to $Q$ do
    Compute the optimization step on $\widetilde{\boldsymbol{w}}$ with learning rate $\alpha^q$ as $\Delta \widetilde{\boldsymbol{w}}^q$;
    Update $\widetilde{\delta \boldsymbol{w}} = \widetilde{\delta \boldsymbol{w}} + \Delta \widetilde{\boldsymbol{w}}^q \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta \boldsymbol{w}}$;
```

complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. Thus, a total extra time complexity is $O(QI + I \cdot \log(I))$ and a total extra space complexity is $O(I)$.

Algorithm 2: Deep Partial Updating. Recall that the dimensionality of the weights vector is denoted as I. In Q optimization iterations during the first step, Alg. 2 introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. The rest of the first step takes a time complexity of $O(I \cdot \log(I))$ and a space complexity of $O(I)$, (*e.g.*, using heap sort or quick sort). In Q optimization iterations during the second step, Alg. 2 introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. Thus, a total extra time complexity is $O(2QI + I \cdot \log(I))$ and a total extra space complexity is $O(I)$.

C Implementation Details

C.1 MLP on MNIST

The MNIST dataset (LeCun & Cortes, 2010) consists of 28×28 gray scale images in 10 digit classes. It contains a training dataset with 60000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 TITAN Xp GPU. We use Adam variant of SGD as the optimizer, and use all default parameters provided by Pytorch. The number of training epochs is chosen as 30, *i.e.*, in the r^{th} round, $Q = |\mathcal{D}^r|/128 \times 30$. The initial learning rate is 0.005, and it decays with a factor of 0.1 every 10 epochs. For fair comparison, we adopt the same learning rate for other baseline methods. Specially, for full updating (FU) and random partial updating (RPU), the network is fully updated and sparsely fine-tuned in $2Q$ iterations, respectively. The corresponding learning rate schedule used in DPU is also proportionally expanded for that used in full updating and random partial updating. That is, the initial learning rate is 0.005, and it decays with a factor of 0.1 for every 20 epochs. This setting is also used in the following experiments. The used MLP contains two hidden layers, and each hidden

layer contains 512 hidden units. The input is a 784-dim tensor of all pixel values for each image. We use ReLU as the activation function, and use a softmax function as the non-linearity of the last layer (*i.e.*, the output layer) in the entire paper. All weights in MLP need around 2.67MB. Each data sample needs 0.784KB. The used MLP architecture is presented as,
2×512FC - 10SVM.

C.2 VGGNet on CIFAR10

The CIFAR10 dataset (Krizhevsky et al., 2009) consists of 32×32 color images in 10 object classes. It contains a training dataset with 50000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 TITAN Xp GPU. We use Adam variant of SGD as the optimizer, and use all default parameters provided by Pytorch. The number of training epochs is chosen as 30, *i.e.*, in the r^{th} round, $Q = |\mathcal{D}^r|/128 \times 30$. The initial learning rate is 0.005, and it decays with a factor of 0.2 every 10 epochs. The used VGGNet is widely adopted in many previous compression works (Courbariaux et al., 2015; Rastegari et al., 2016), which is a modified version of the original VGG (Simonyan & Zisserman, 2015). All weights in VGGNet need around 56.09MB. Each data sample needs 3.072KB. The used VGGNet architecture is presented as,
2×128C3 - MP2 - 2×256C3 - MP2 - 2×512C3 - MP2 - 2×1024FC - 10SVM.

C.3 ResNet34 on ILSVRC12

The ILSVRC12 (ImageNet) dataset (Russakovsky et al., 2015) consists of high-resolution color images in 1000 object classes. It contains a training dataset with 1.2 million data samples, and a validation dataset with 50000 data samples. Following the commonly used pre-processing (Pytorch, 2019), each sample (single image) is randomly resized and cropped into a 224×224 color image. We use the original training dataset for training; and randomly select 15000 samples in the original validation dataset for validation, and the rest 35000 samples for testing. We use a mini-batch with size of 512 training on 4 TITAN Xp GPUs. According to the suggestions in (Renda et al., 2020), we use Nesterov SGD as the optimizer. We also use the parameters provided by (Renda et al., 2020) to configure Nesterov SGD optimizer. The number of training epochs is chosen as 45, *i.e.*, in the r^{th} round, $Q = |\mathcal{D}^r|/512 \times 45$. Thus, the number of training epochs for full updating is 90 as in (Renda et al., 2020). The learning rate schedule in (Renda et al., 2020) is shrunk proportionally to fit in 45 epochs. The ResNet34 used in our experiments is proposed in (He et al., 2016). All weights in ResNet34 need around 87.12MB. Each data sample needs 150.528KB. The network architecture is the same as "resnet34" in (Pytorch, 2019).

C.4 LSTM on Penn Treebank

We conduct partial updating on the word-level language modeling task to study the performance of DPU on recurrent neural network architectures. We use one of the most well-known recurrent neural networks, *i.e.*, long short-term memory (LSTM) (Hochreiter & Schmidhuber, 1997). We choose Penn Treebank (PTB) corpus (Marcus et al., 1993) as the evaluation benchmark. The PTB dataset contains 929K training tokens, 73K validation tokens, and 82K testing tokens, where one token corresponds to one word. We still use the average cross-entropy as the loss function, and use Adam variant of SGD as the optimizer with a clipping on the gradient norm at 0.5 (Choi, 2020). Since language modeling aims at predicting the next word, the performance is measured by perplexity per word (PPW) metric. The lower the perplexity, the higher the model performance. The test perplexity is reported, when the validation dataset achieves the lowest perplexity.

We mainly follow all settings in (Choi, 2020). We use the standard pre-processing splits with a 10K size vocabulary. The file content is tokenized with a mini-batch size of 20, *i.e.*, the file is sequentially and uniformly split into 20 arrays. Since the tokenized file is constructed sequentially, both $\mathcal{D}^1$ and $\delta\mathcal{D}^r$ are cropped from 20 arrays sequentially, with $|\mathcal{D}^1|$ and $|\delta\mathcal{D}^r|$ tokens from each of 20 arrays respectively (only for evaluation purposes). The newly collected $\delta\mathcal{D}^r$ is also sequentially concatenated with the previous dataset to build the new training dataset at each round. For example, $\mathcal{D}^2 = \mathcal{D}^1 \cup \delta\mathcal{D}^2$, the total number of tokens contained in $\mathcal{D}^2$ is $|\mathcal{D}^2| \times 20 = (|\mathcal{D}^1| + |\delta\mathcal{D}^2|) \times 20$. Let $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ represent the available data samples along rounds, where $|\delta\mathcal{D}^r|$ is supposed to be constant along rounds. In this case, each data sample corresponds to 20 tokens (words). We use a fixed learning rate of 0.002. The used LSTM is also the same as (Choi, 2020), which contains 1 hidden layer of size 1024 and an embedding layer of size 128. We unroll the network for 30 time steps, *i.e.*, the sequence length is 30. The number of training epochs is chosen as 5 according to (Choi, 2020), *i.e.*, in the r^{th} round, $Q = (\text{ceil}(|\mathcal{D}^r|/30) - 1) \times 5$. All weights in LSTM need around 64.95MB. Since each English word contains 5 characters in average, each data sample (20 words) needs around 120 characters including the space, *i.e.*, 0.120KB in ASCII format.

D Additional Experimental Results

D.1 Full Updating

Settings. In this experiment, we compare full updating with different initialization methods at each round in terms of the test accuracy. The compared full updating methods include, (*i*) the network is trained from a random initialization at each round; (*ii*) the network is trained from a same random initialization at each round, *i.e.*, with a same random seed; (*iii*) the network is trained from the weights $\boldsymbol{w}^{r-1}$ of the last round at each round. For fair comparison, all methods are trained with the same learning rate schedule and the same number of training iterations per round. Since training from a random initialization (*i.e.*, training from scratch) always requires more iterations (epochs) than training from the last round, the number of training iterations is selected to ensure all methods can be fully optimized within these iterations. The corresponding number of training iterations used at each round is set to $2Q$. The experiments are conducted on VGGNet using CIFAR10 dataset with different amounts of training samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$. Each experiment runs for three times using random data samples and different random seeds.

Results. We report the mean and the standard deviation of test accuracy (over three runs) under different initialization in Figure 4. The results show that training from a same random initialization yields a similar accuracy level while sometimes also a lower variance, as training from a (different) random initialization at each round. In comparison to training from scratch (*i.e.*, random initialization), training from $\boldsymbol{w}^{r-1}$ may yield a higher accuracy in the first few rounds; yet training from scratch may always outperform after a large number of rounds. Thus, in this paper, we adopt training from a same random initialization at each round, *i.e.*, (*ii*), as the baseline of full updating.

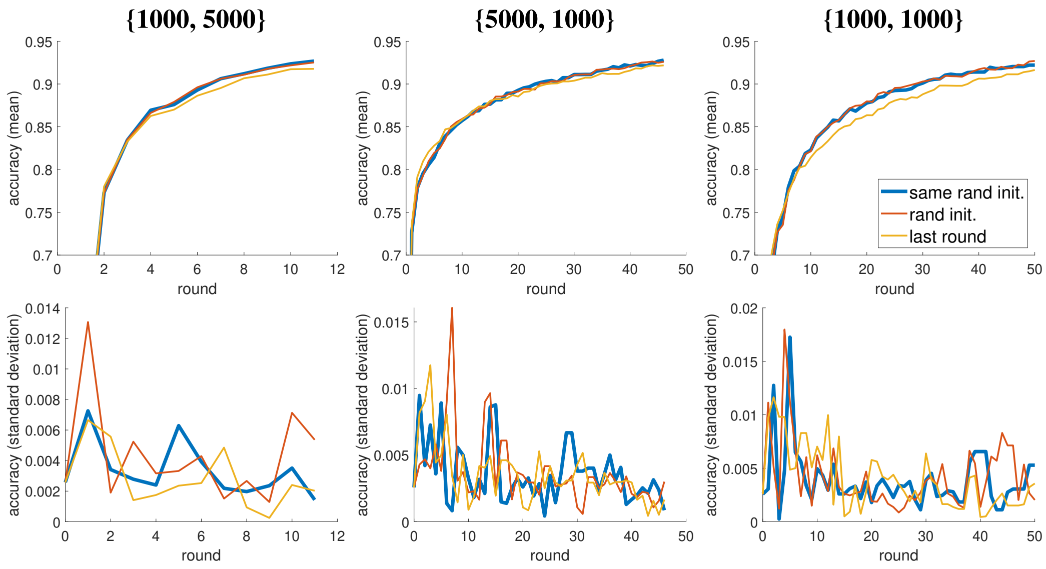

Figure 4: Comparing full updating methods with different initialization methods at each round.

D.2 Number of Rounds for Re-Initialization

Settings. In these experiments, we re-initialize the network every n rounds under different partial updating settings to determine a heuristic rule for setting n. We conduct experiments on VGGNet using CIFAR10 dataset, with different amounts of training samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and different updating ratios k. Every n rounds, the network is (re-)initialized again from a same random network (as mentioned in D.1), then partially updated in the next n rounds with Alg. 2. We choose $n = 1, 5, 10, 20$. Specially, $n = 1$ means that the network is partially updated from the same random network every round. Each experiment runs three times using random data samples.

Results. We plot the mean and the standard deviation of the test accuracy along rounds in Figure 5 and Figure 6, respectively. By comparing $n = 1$ with other settings, we can conclude that within a certain number of rounds, the current deployed network $\boldsymbol{w}^{r-1}$ (*i.e.*, the network from the last round) is a better starting point for Alg. 2 than a randomly initialized network, *i.e.*, partially updating from the last round may yield a higher accuracy than partially updating from a random network with the same training effort. This is straightforward, since such a network is already pretrained on a subset of the currently available data samples, and the previous learned knowledge could help in the new training process. Since any newly collected samples are continuously stored in the server, complete information about all past data samples is available. This also makes our setting different from continual learning setting, which aims at avoiding catastrophic forgetting without accessing (at least not all) old data.

Each time the network is re-initialized, the new partially updated network may suffer from an accuracy drop. Although this accuracy drop may be relieved if we carefully tune the partial updating training scheme every time, this is not feasible regarding the large number of updating rounds. However, we can simply avoid such an accuracy drop by not updating the network if the validation accuracy does not increase compared to the last round (as discussed in Section 4). Note that in this situation, the partially updated weights (as well as the random seed for re-initialization) still need to be sent to the edge devices, since this is an on-going training process.

After implementing the above strategy, we plot the mean accuracy in Figure 7. In addition, we also add the related results on full updating in Figure 7, where the network is re-initialized every n rounds from a same random network. Note that full updating with re-initialization every round ($n = 1$) is exactly the same as "same rand init." in Figure 4 in D.1. From Figure 7, we can conclude that the network needs to be re-initialized more frequently in the first several rounds than in the following rounds to achieve a higher accuracy level. The network also needs to be re-initialized more frequently with a large partial updating ratio k. Particularly, the ratio between the number of current data samples and the number of following collected data samples has a larger impact than the updating ratio.

Thus, we propose to re-initialize the network as long as the number of total newly collected data samples exceeds the number of samples when the network is re-initialized last time. For example, assume that at round r the model is randomly (re-)initialized and partially updated from this random network on dataset $\mathcal{D}^r$. Then, the model will be re-initialized at round $r + n$, if $|\mathcal{D}^{r+n}| > 2 \cdot |\mathcal{D}^r|$.

D.3 Evaluation on Different Benchmarks

D.3.1 Experiments on Total Communication Cost Reduction

Settings. In this experiment, we show the advantages of DPU in terms of the total communication cost reduction, as DPU has no impact on the edge-to-server communication which may involve sending newly collected data samples on nodes. The total communication cost includes both edge-to-server communication and server-to-edge communication. Here we assume that all samples in $\delta\mathcal{D}^r$ are collected by N edge nodes during all rounds and sent to the server on a per-round basis. For clarity, let S_d denote the data size of each training sample. During round r, we define the per-node total communication cost under DPU as $S_d \cdot |\delta\mathcal{D}^r|/N + (S_w \cdot k \cdot I + S_x(k) \cdot I)$. Similarly, the per-node total communication cost under full updating is defined as $S_d \cdot |\delta\mathcal{D}^r|/N + S_w \cdot I$.

In order to simplify the demonstration, we consider the scenario where N nodes send a certain amount of data samples to the server in $R - 1$ rounds, namely $\sum_{r=2}^{R} |\delta\mathcal{D}^r|$ (see Section 3.2). Thus,

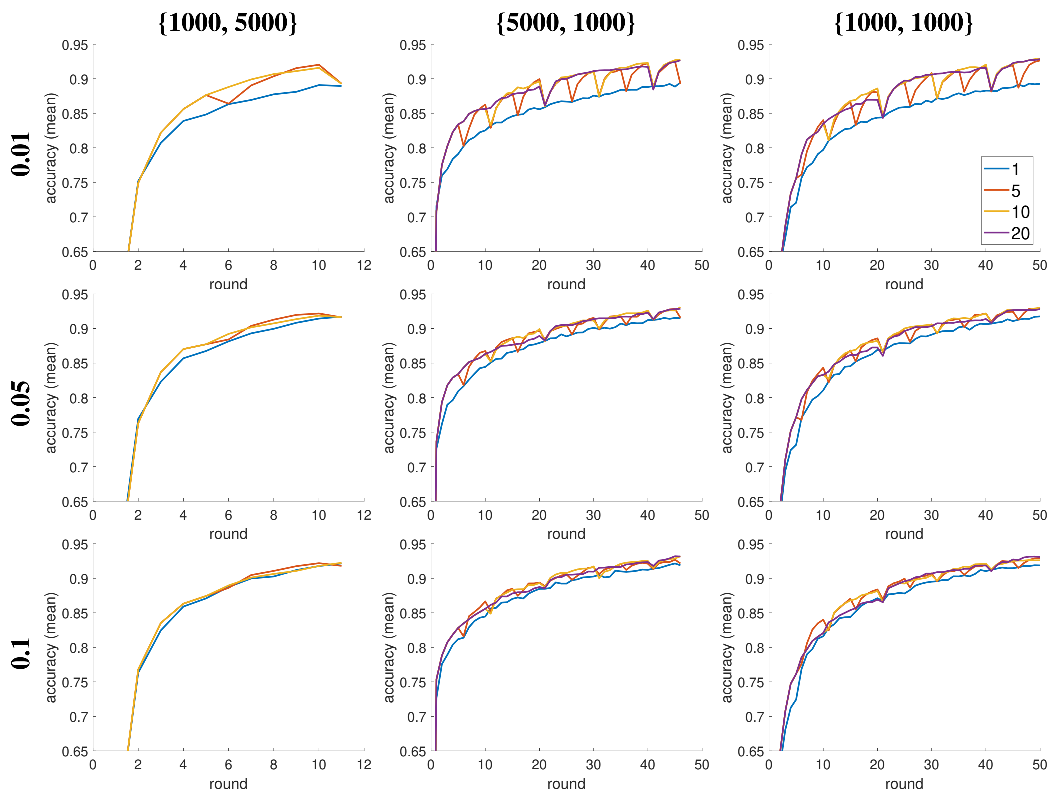

Figure 5: Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

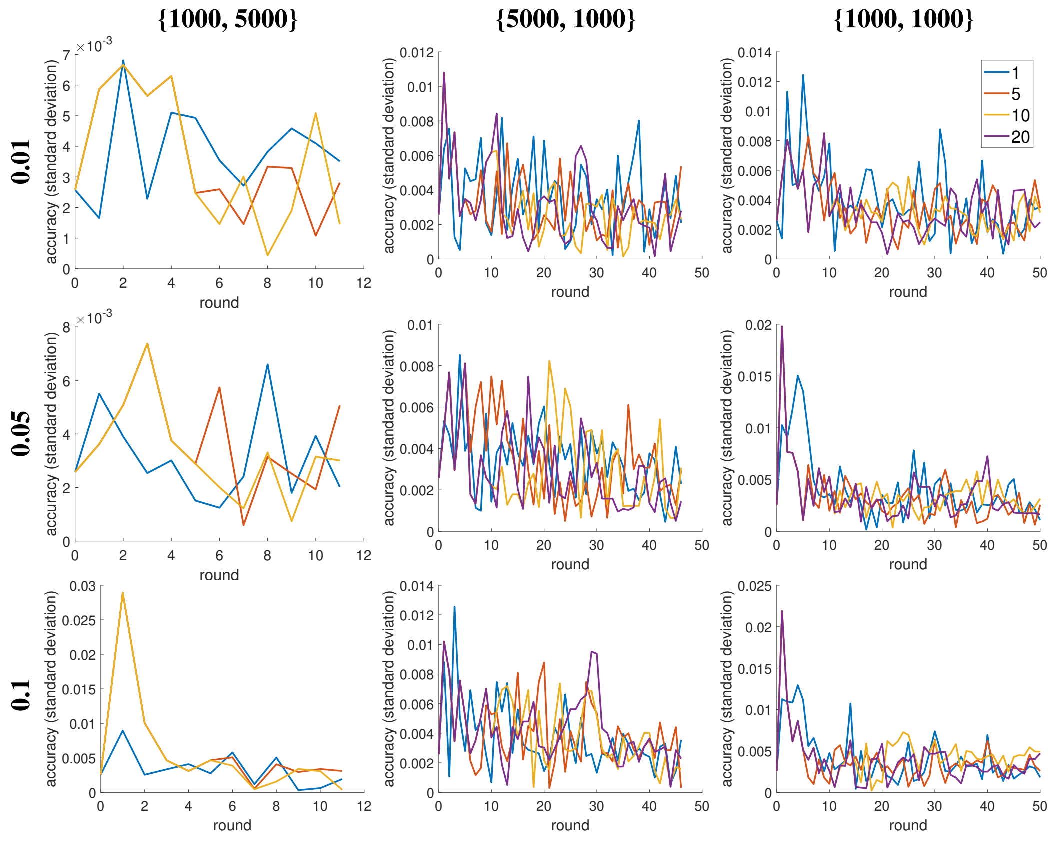

Figure 6: Comparison w.r.t. the standard deviation of accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

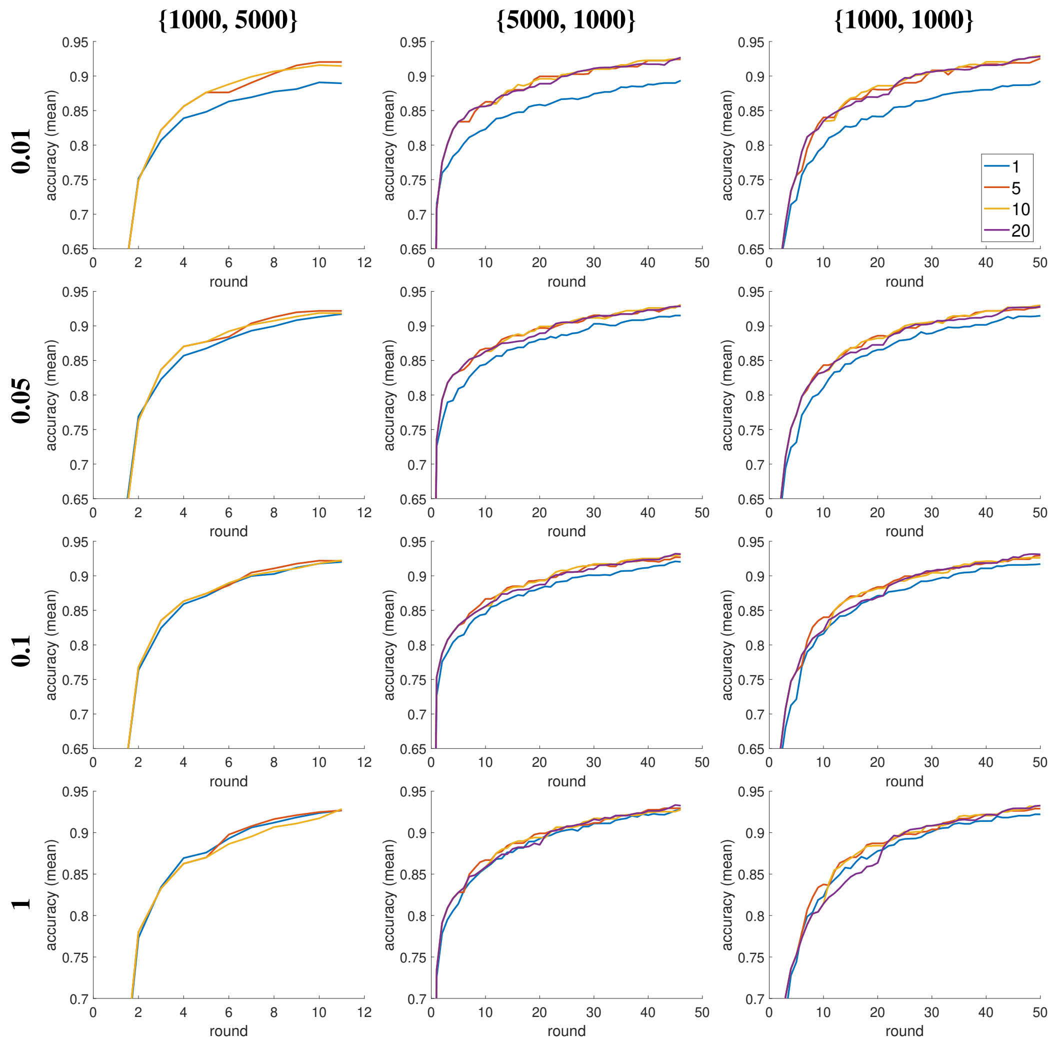

Figure 7: Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$ and full updating $k = 1$) settings.

the average data size transmitted from each node to the server in all rounds is $\sum_{r=2}^{R} S_d \cdot |\delta \mathcal{D}^r|/N$. A larger N implies a fewer amount of transmitted data from each node to the server.

Results. We report the ratio of the total communication cost over all rounds required by DPU related to full updating, when DPU achieves a similar accuracy level as full updating (corresponding to three evaluations in Figure 2). The ratio clearly depends on $\sum_{r=2}^{R} S_d \cdot |\delta \mathcal{D}^r|/N$, *i.e.*, the number of nodes N. The relation between the ratio and N is plotted in Figure 8.

DPU can reduce up to 88.2% of the total communication cost on updating MLP and VGGNet even for only a single node. Single node corresponds to the largest data size during edge-to-serve transmission per node, *i.e.*, the worst case. Moreover, DPU tends to be more beneficial when the size of data transmitted by each node to the server becomes smaller. This is intuitive because in this case the server-to-edge communication cost (thus the reduction due to DPU) dominants in the entire communication cost. For tasks with a large S_d, DPU can still significantly save the total communication cost with a large number of nodes (*e.g.*, some mobile applications). For example, partial updating ResNet34 on ILSVRC12 can save over 50% of the total communication cost in a 500-node sensing system.

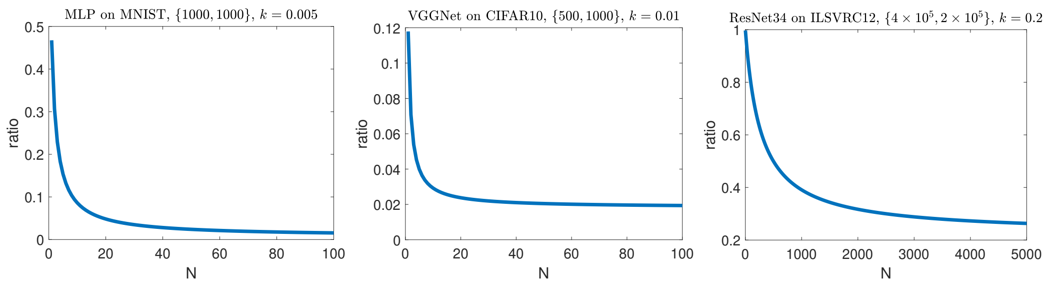

Figure 8: The ratio, between the total communication cost (over all rounds) under DPU and that under full updating, varies with the number of nodes N.

D.3.2 LSTM on Penn Treebank

Settings. Similar as Section 4.2, we also compare DPU to three baseline methods on LSTM using Penn Treebank dataset, including (*i*) full updating (FU), where at each round the network is fully updated with a random initialization; (*ii*) random partial updating (RPU), where the network is trained from $\boldsymbol{w}^{r-1}$, while we randomly fix each layer's weights with a ratio of $(1-k)$ and sparsely fine-tune the rest; and (*iii*) global contribution partial updating (GCPU), where the network is trained with Alg. 1 without re-initialization described in Section 3.2. We conduct the experiment under three updating ratios, 0.005, 0.01, and 0.05.

Results. We plot results in terms of test perplexity in Figure 9. As seen in this figure, DPU clearly yields the lowest perplexity in comparison to the other partial updating schemes on LSTM. Similar as Section 4.2, we compare three partial updating schemes in terms of the perplexity difference related to full updating averaged over all rounds, as well as the ratio of the communication cost (server-to-edge) over all rounds related to full updating. We report the results under updating ratio $k = 0.05$ in Table 3. As seen in the table, DPU reaches a similar perplexity as full updating, while incurring significantly fewer transmitted data sent from the server to each edge node. In addition, we also report the ratio of the total communication cost (including edge-to-server and server-to-edge) over all rounds required by DPU related to full updating in Figure 10. Note that text data samples require a relatively smaller data size compared to image data samples. As seen in this figure, DPU always achieves a significant reduction on the total communication cost (*e.g.*, 94% reduction even for the most pessimistic scenario, *i.e.*, $N = 1$).

D.4 Impact due to Varying Number of Data Samples and Updating Ratios

Settings. In this set of experiments, we demonstrate that DPU outperforms other baselines under varying number of training samples and updating ratios. We also conduct an ablation study concerning

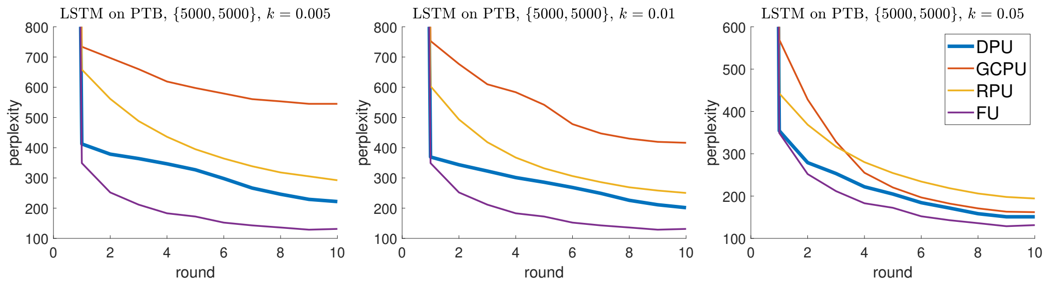

Figure 9: DPU is compared with other baselines on LSTM using PTB dataset in terms of the test perplexity.

Table 3: The average perplexity difference over all rounds and the ratio of communication cost (server-to-edge) over all rounds related to full updating (with $k = 0.05$).

Method	Average perplexity difference	Ratio of communication cost
DPU	$\mathbf{+27.0}$	0.0531
GCPU	$+81.7$	0.0589
RPU	$+85.4$	0.0589

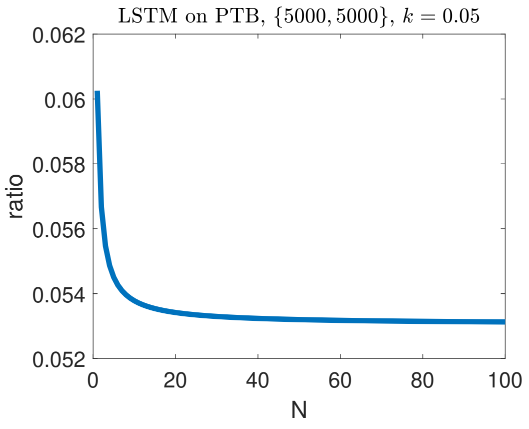

Figure 10: The ratio, between the total communication cost (over all rounds) under DPU and that under full updating, varies with the number of nodes N.

the re-initialization of weights discussed in Section 3.2. We implement DPU with and without re-initialization, GCPU with and without re-initialization and RPU (see Section 4.2) on VGGNet using CIFAR10 dataset. We compare these methods with different amounts of samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and different updating ratios k. Each experiment runs three times using random data samples.

Results. We compare the difference between the accuracy under each partial updating method and that under full updating. The mean accuracy difference (over three runs) is plotted in Figure 11. The standard deviation of the accuracy difference (over three runs) is provided in Figure 12. As seen in Figure 11, DPU (with re-initialization) always achieves the highest accuracy. In addition, we also plot the mean and standard deviation of test accuracy (over three runs) of these methods (including full updating) in Figure 13 and Figure 14, respectively. The dashed curves and the solid curves with the same color can be viewed as the ablation study of our re-initialization scheme. Particularly given a large number of rounds, it is critical to re-initialize the start point $\boldsymbol{w}^{r-1}$ after performing several rounds (as discussed in Section 3.2).

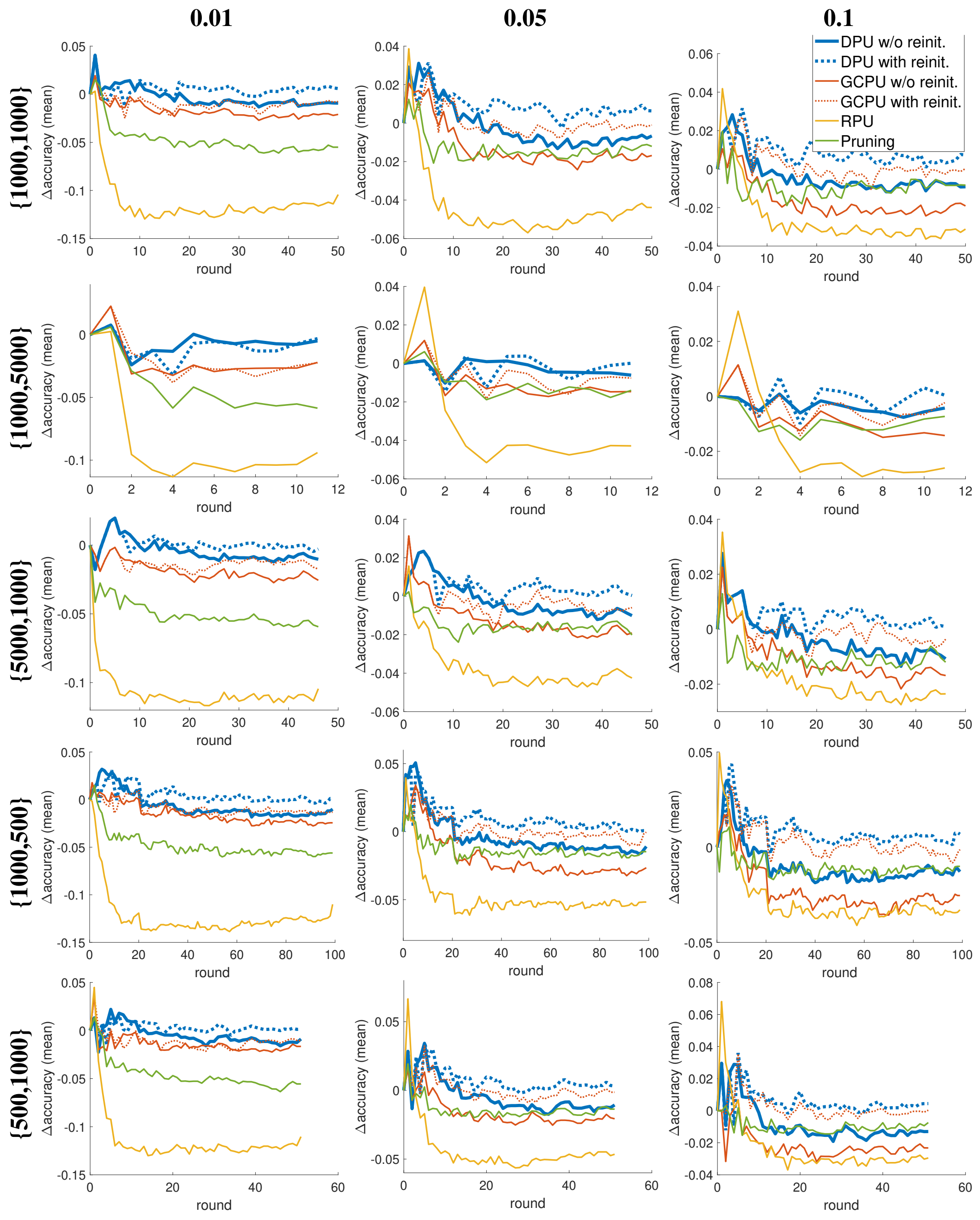

Figure 11: Comparison w.r.t. the mean accuracy difference (full updating as the reference) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

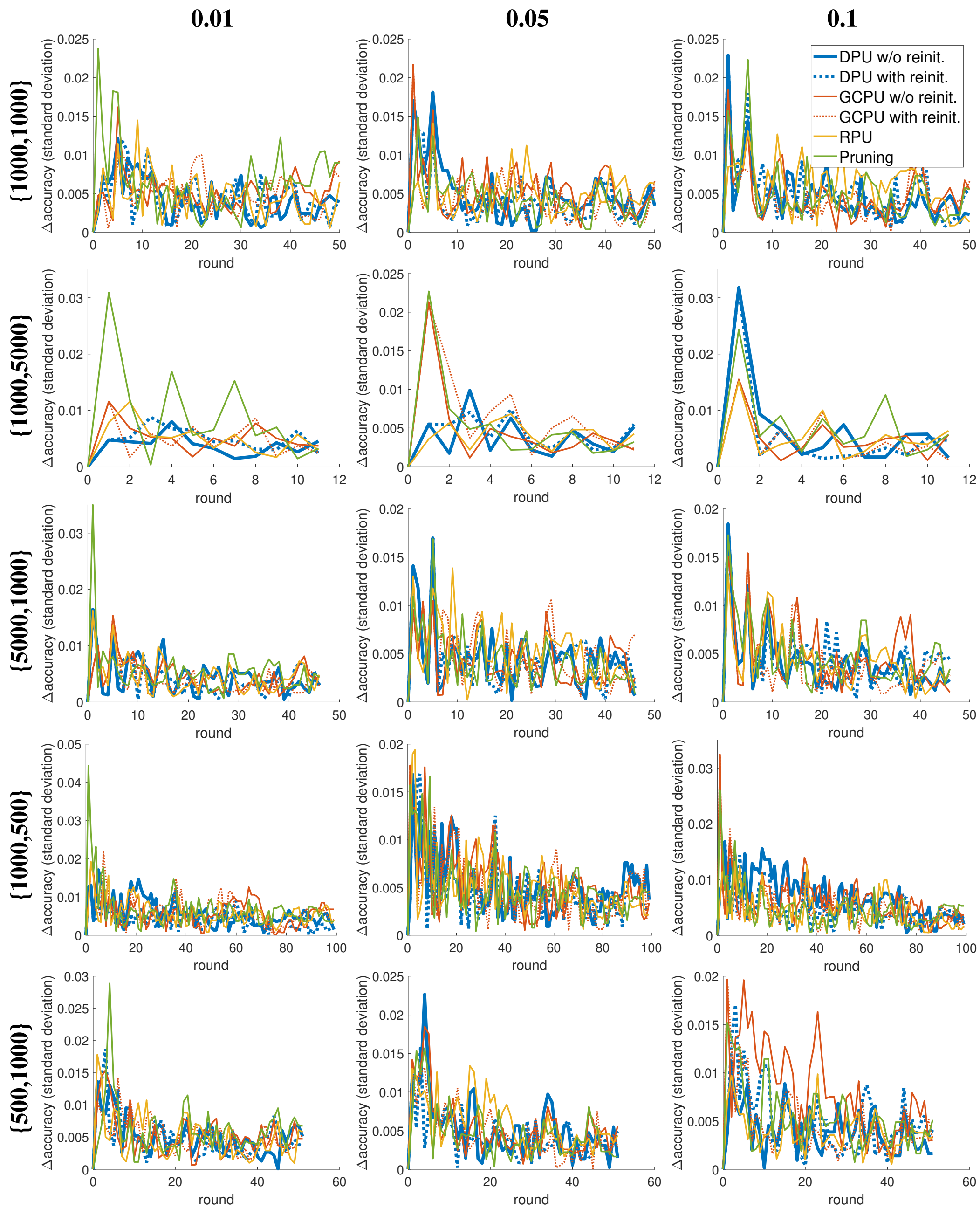

Figure 12: Comparison w.r.t. the standard deviation of accuracy difference (full updating as the reference) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

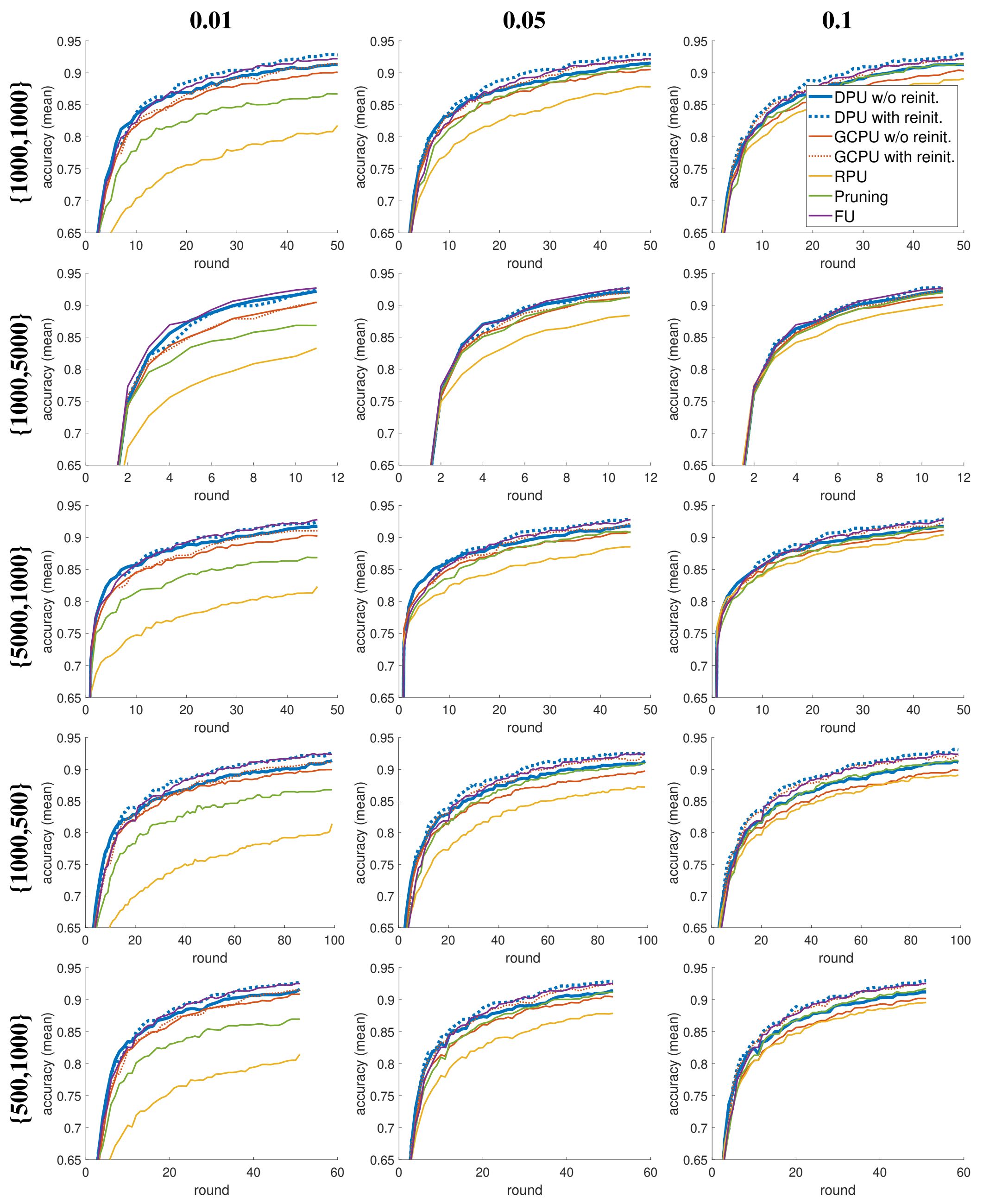

Figure 13: Comparison w.r.t. the mean accuracy under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

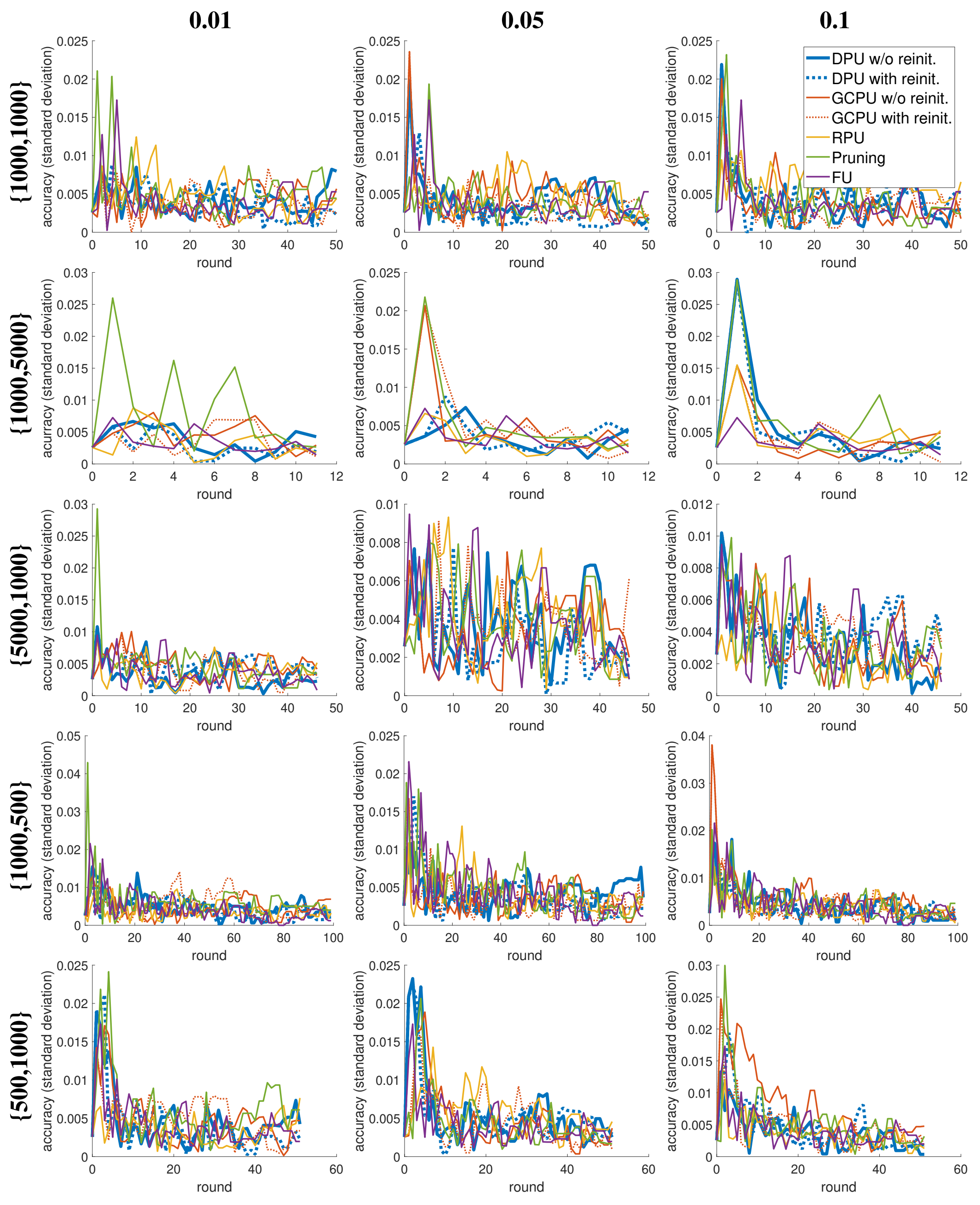

Figure 14: Comparison w.r.t. the standard deviation of accuracy under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.